711-61-CR

210985

16P

ICASE REPORT

# A LANGUAGE DESIGN FOR VECTOR MACHINES

V. R. Basili

J. C. Knight

Report Number 75-3
February 18, 1975

INSTITUTE FOR COMPUTER APPLICATIONS

IN SCIENCE AND ENGINEERING

Operated by the

UNIVERSITIES SPACE RESEARCH ASSOCIATION

at

NASA's LANGLEY RESEARCH CENTER

Hampton, Virginia

(NASA-CR-179994) A LANGUAGE DESIGN FOR VECTOR MACHINES (ICASE) 16 p

N89-71118

Unclas
00/61 0210985

# A LANGUAGE DESIGN FOR VECTOR MACHINES

V. R. Basili
Dept. of Computer Science
University of Maryland
College Park, MD 20742

J. C. Knight
Analysis and Computation Division
NASA Langley Research Center
Hampton, VA 23665

## ABSTRACT

This paper deals with a programming language under development at NASA's Langley Research Center for the CDC STAR-100. The design goals for the language are that it be basic in design and able to be extended as deemed necessary to serve the user community, capable of the expression of efficient algorithms by forcing the user to make the maximum use of the specialized hardware design, and easy to implement so that the language and compiler could be developed with a minimum of effort. The key to the language was in choosing the basic data types and data structures. Scalars, vectors, and strings are available data types in the language. Each basic data type has an associated set of operators which consist primarily of the operations provided by the hardware. The only data structure in the language is a restricted form of the array. Only vector and string data types may be stored in arrays, forcing the user to vectorize scalar data when it is necessary to structure it. This permits the most effective use of the machine for entities such as real arrays since the high level vector machine instructions may be used to deal with them directly.

This paper is a result of work started under NASA Grant NGR 47-102-001 while the authors were in residence at ICASE, NASA Langley Research Center.

## INTRODUCTION

Vector and parallel processing machines offer new problems in the area of language design. Besides the goal of designing the language which is best suited to the user for his particular application, there is the added problem of making effective use of the specialized architecture. The relatively high level nature of the machine plays an important role in the level of the languages designed for it.

In general, there are three approaches that might be examined for these machines. First, an existing sequentially oriented language, such as FORTRAN, may be expanded in an attempt to handle the new vector or parallel capabilities. Second, a very high level language relative to the new hardware may be designed or adapted. The third choice is a language which is intentionally influenced by the characteristics of the hardware in what amounts to a "bottom up" approach to language design.

This paper deals with the design and motivation for a programming language, presently called SL/1, under development at NASA's Langley Research Center for the CDC STAR-100. The design goals for the language are that it be:

1. basic in design. That is, language features were included only if they were felt to be absolutely necessary,
2. able to be extended as deemed necessary to serve the user community,
3. capable of expressing efficiently executable algorithms by forcing the user to make maximum use of the specialized hardware design,
4. easy to implement so that the language and compiler could be developed with a minimum of effort.

For these reasons, the level of the language is vector architecture dependent and therefore directed at the level of the machine; i.e., the third choice.

## DESIGN JUSTIFICATION

Some justifications will be given on why this approach satisfied the design goals. At the lower level, a sequentially oriented language could be extended by incorporating some form of vector processing capability. This is the approach being taken by CDC in the development of their FORTRAN compiler for the STAR-100. However, these extensions inevitably take on the appearance of patches to the basic language design and valuable support constructs, such as appropriate data and control structures, are usually missing. This can leave the language with an inconsistent and cumbersome design.

Certainly, an implementation of FORTRAN, perhaps with extensions, is needed to allow existing programs to run on the new machine. FORTRAN has the benefit of being well known and used by the relevant programming community. However, this familiarity is also a drawback since the user is in the habit of writing algorithms in FORTRAN which are centered around the manipulation of scalar quantities. For a machine like the STAR, this is not the appropriate level of algorithm expression for making best use of the hardware. Recognizing the equivalence of sequentially written algorithms to a single machine instruction is impossible in the general case, and is often inefficient in those special cases that are worth detecting. For example, the following piece of FORTRAN evaluates a polynomial with a set of coefficients A and for a set of argument values X:

```
          LIMIT = N+1
          DO 10 J = 1,M
          VALUE(J) = X(J)*A(1)
          DO 10 I = 2,LIMIT
      10  VALUE(J) = (VALUE(J)+A(I)*X(J)
```

This is equivalent to one machine instruction on the STAR. Recognition of this fact by special case is quite difficult and costly.

Thus, the implementation of a compiler for the language is a major effort since it requires the translation of all of FORTRAN, the new special features, along with any number of special recognition cases that are to be included.

The choice of a language which is much higher than the level of the machine is theoretically a better choice. An existing language like SETL [1] could be chosen, or a new language could be designed. Relative to the design goals, the very high level nature of the language would most likely eliminate the need for extensions. It would certainly permit the easy expression of high level algorithms theoretically suitable for efficient execution on the machine. The practical implementation of the optimization techniques is another question.

The primary drawback to this high level of language, however, would be the major design and implementation effort involved. Unless an existing language like SETL is chosen, the language design effort alone is an extensive research undertaking. In either case, the implementation of such a language is beyond the resources available to the project. In addition, compile time for programs written in the language would be inordinately high, and there are some strong feelings in the user community against programming at so high a level because of the lack of user control over run time efficiency.

SL/1 is designed at a level that capitalizes on vector architecture, and corresponds closely to the level of the machine. In designing a language at the machine level, care must be given to specifying a consistent, easy to use, reliable language which makes use of the power of the hardware without exposing the user to hardware idiosyncrasies. The purpose of this design effort is not a high level assembly language, but a high level algorithmic language that would provide the user with the appropriate set of data and control structures for expressing algorithms in a readable and efficiently executable form.

The language design is relatively simple, which makes it easy to extend. The basic conservative language design was motivated partially by the design for the base language SIMPL_T [2] of the SIMPL family of programming languages [3]. SL/1 is meant to be a base language for a possible family of languages, each of which would serve a special application area of the user community. Each language could be designed as an extension to the base language and the compiler built as an extension to the base compiler.

It is difficult to specify a consistent design level for a bi-level (both scalar and vector instructions) machine like the STAR. However, the specification of basic data types and data structures in SL/1 solve both the consistent level problem and the problem of forcing the user into making maximum use of the machine hardware. Scalars (real, short real, integer, character,...), vectors (real vector, short real vector, integer vector,...), and strings (character string, bit string,...) are available data types in the language. The only data structure in the language is the array, and only vector and string data types may be stored in arrays. Thus, the user is forced to vectorize scalar data when it is necessary to structure it. This permits the most effective use of the machine for entities such as real arrays since the high level vector machine instructions may be used to deal with them directly.

## BASIC DESIGN

### Data Types and Operators

The STAR hardware is capable of operating on scalars, vectors, and strings. Scalars include 32- and 64-bit quantities (ostensibly floating point numbers) with arithmetic, logical, and relational operations. Vectors are of two types: normal and sparse. Normal vectors consist of a sequence of 32- or 64-bit quantities occupying contiguous storage locations. Sparse vectors are described by a sequence of nonzero elements and an associated characteristic vector (bit pattern). There are sets of high level hardware operations for both of these vector types. Strings on the STAR are similar to vectors except that they consist of sequences of bits or bytes of information for which there is no scalar equivalent. They also have a set of high level hardware operators associated with them.

In SL/1, an attempt was made to define data types into a more organized and complete classification scheme, and to provide the user with a more unified and specified set of data elements. Each data type may not have an exact counterpart in the STAR hardware, but it is usually easily simulated. The scalar quantities are divided into six types. They are:

| | | |
|---|---|---|
| (a) | Real | - 64-bit floating point |
| (b) | Short Real | - 32-bit floating point |
| (c) | Integer | - 48-bit integer |
| (d) | Short Integer | - 24-bit integer |
| (e) | Logical | - single bit |
| (f) | Character | - 8-bit byte |

The integer quantities are just floating point numbers with zero exponents.

Vector data types in SL/1 are defined as fixed length one-dimensional arrays consisting of elements of a specified scalar type. Present vector types include:

(g) Real vector

(h) Short real vector

(i) Integer vector

(j) Short integer vector

(k) Logical vector

Although the hardware is also able to deal with sparse vectors, they are not included in the first version of SL/1. In the interest of simplicity, sparse vectors will not be considered until the first language extension.

In contrast to vectors, strings in SL/1 are defined to be execution time variable sequences of a specified scalar type. Present string data types include only:

(ℓ) Character string

Vectors and strings may be declared with the CONTROLLED attribute in which case storage can be allocated and freed at run time.

The set of operators available in the language is considerable. It essentially includes most of those which the hardware can deal with directly plus whatever minimal extensions were necessary to handle the new data types. One approach to the syntax is to define a single symbol for each operator as has been done in APL [4]. However, the restriction imposed by available character sets makes this impractical. Where no obvious symbol exists, dyadic SL/1 operators consist of a meaningful sequence of letters with a period as prefix and suffix, as in FORTRAN, and monadic and triadic operators are written as function calls. For example, the polynomial evaluation written in FORTRAN above is written in SL/1 as:

```
VALUE :=A .EVL. X ;
```

where, as before, VALUE is the vector of results, A is the vector of coefficients, and X is the vector of argument values. An extensive list of the vector operators is included in Appendix 2.

All declarations in SL/1 are explicit and are similar to the standard Algol-like format; e.g.,

```
REAL VECTOR  [1..10]  A;
     /* Declaration of a vector with ten real elements with
     subscript range from 1 to 10*/
CHARACTER STRING  [100] B;
     /* Declaration of a character string with a maximum length
     of one hundred characters */
```

```
CONTROLLED REAL VECTOR  [1001..11000]  C;
          /* Declaration of a vector with 10,000 real elements for
          which no storage is reserved */
```

Two aspects of the language of particular interest are the referencing of vectors and strings, and the syntax of vector and string constants. A single element of a vector or string is referenced merely by specifying the required index in the normal way; e.g., A[5] := 1; B[2] := 'C';. The STAR-100 hardware has the useful facility of permitting a reference to a vector or string to be offset, so that a given instruction may begin processing a vector or string at some point other than its beginning. In addition, the number of elements to be processed may be set as a length. This capability is used in SL/1 to allow subvectors and substrings to be specified as a first element, last element pair, or a first element, length pair. For example, to reference elements 12 through 16 inclusive of a vector V, the syntax is V[12..16] or V[12!5]. Both notations are provided because of their frequent occurrence in the mathematical statement of algorithms. A similar substring notation is used for strings.

Vector constants in SL/1 are element sequences which can be written out in full. For example, \1,2,3,4\ is a four element vector constant. String constants are treated exactly as they are in SNOBOL. More complex constant vectors and strings can be created using the normal operators of the language, such as replication and concatenation.

## Data Structures

The only data structure provided by SL/1 is the array with the restriction that each element can only be a vector or string. The purpose of this restriction is to ensure that the user structures information as vectors rather than declaring arrays of scalars and attempt to use them as one would in a traditional programming language. This requirement forces the handling of linear sequences of data in a more efficient manner.

A one-dimensional array of vectors or "vector array" is similar in nature to a matrix. The problem of providing a matrix or any multidimensional data structure is the inherent symmetry of the indices. For example, the user tends to regard referencing rows and columns of a matrix as equivalent. If a matrix is stored rowwise on the STAR-100, then any operations on the rows can use the machine's vector processing capability directly. However, column operations are faced with tremendous overhead since the elements of a column do not occupy sequential storage locations. Any programming language which provides a multidimensional array capability with scalar elements must also provide facilities for user control over how the array is stored, and warn the user when his references to the array are inefficient.

The vector array avoids these problems. It is the user's responsibility to interpret the vectors in the array as he wishes. For example, if each element of a one-dimensional array has been used to store a row of a matrix, then row operations are easily programmed and efficiently implemented. However, to access a column, the user must explicitly program the element by element reference pattern and the associated inefficiency is clear.

A common occurrence in scientific programming is the triangular system, and it is usually left to the programmer's ingenuity to ensure that it is efficiently stored. Since the STAR-100 will be used primarily for scientific computing, SL/1 allows vector arrays to be declared such that the lengths of the element vectors form an arithmetic progression. By making both the length of the first vector and the increment one, a triangular system can be stored.

The declaration of a vector array in which all the element vectors are of the same length consists of declaration information for the element vectors, followed by the word ARRAY followed by the array dimensional information contained in parenthesis; e.g.,

```
REAL VECTOR  [101..300] ARRAY  (1..10) X;
          /*  X consists of ten vectors, each of which is two hundred
          elements long. */
REAL VECTOR  [51..100] ARRAY  (1..20, 1..10) Y;
          /*  Y consists of a 20x10 array, each element of which is
          a vector with 50 elements.  */
```

The entire vector which is the ith element of X may be referenced using the notation X(i). The jth element of that particular vector may be referenced by writing X(i)[j]. Similarly, the vector which is the i,j element of Y may be referenced as Y(i,j).

A triangular system is declared similarly, but the first two parameters within the brackets define only the first element vector. A third parameter is used to specify the length difference between adjacent element vectors; e.g.,

```
REAL VECTOR  [1..1 BY 1]  ARRAY (1..10) Y;
          /*  Y is a triangular system consisting of ten vectors,
          the first of length one, and the i+1st vector one element
          longer than the ith.  Thus, for Y, the ith element vector
          is of length i.*/
```

If the first vector length is greater than one, a negative difference may be specified indicating decreasing vector lengths.

## Statements

In an SL/1 assignment statement, the right hand side may produce a scalar, string, or vector value. If a scalar or string is the result, assignment takes place in the normal way. However, when the right hand side yeilds a vector, the semantics of the operator are more complex because of unique features of the STAR-100 vector hardware.

Many of the STAR vector instructions allow storage of the result vector to be controlled by a bit vector. If a bit vector is used, then the $i^{th}$ result element is stored if the $i^{th}$ bit is one; otherwise, it is discarded. The hardware also allows the opposite operation; i.e., store on zero, discard on ones.

This feature of vector assignment essentially makes the assignment operator triadic. In SL/1, the result variable and bit vector are both written on the left hand side of the assignment operator. They are separated by a comma and surrounded by parentheses; e.g.,

```
(C,Z)  := A+B;
```

In this example, C is the result field and Z is the bit vector used to control the store operations.

Most of the commonly occurring control statements are available in SL/1. For example, the WHILE, REPEAT UNTIL, CASE, FOR, and IF statements are provided. Compound statements are bracketed by language keywords wherever possible rather than BEGIN and END. The recent practice of using a keyword spelt backwards (FI, ESAC, etc.) to delimit the end of a statement has not been followed since the authors feel that this can be confusing. Instead, the word END is used with a single letter suffix added to indicate the type of statement. For example, the full form of the IF statement is:

```
IF <Boolean Expression> THEN <Statement List>
                        ELSE <Statement List> ENDI
```

## CONCLUSION

At this writing, the design of SL/1 is in the final stages of refinement. Potential STAR users have been actively involved in the design process by programming algorithms in the language, and by giving feedback on language constructs.

Three typical SL/1 program segments are included in Appendix 1 as examples.

## ACKNOWLEDGMENT

Several people have contributed to the refinements of the design of SL/1. The authors would like to express their appreciation in particular to Edmond H. Senn and Rudeen S. Smith of NASA's Langley Research Center, and to Michael Donegan and Stuart Katzke of the College of William and Mary.

## REFERENCES

1. Schwartz, J.: "On Programming: An Interim Report On The SETL Project," Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1973.

2. Basili, V. R. and Turner, A. J.: "SIMPL_T: A Structured Programming Language," Computer Note - CN14, Computer Science Center, University of Maryland, 1974.

3. Basili, V. R.: "The SIMPL Family of Programming Languages and Compilers," Technical Report - TR305, Computer Science Center, University of Maryland, 1974.

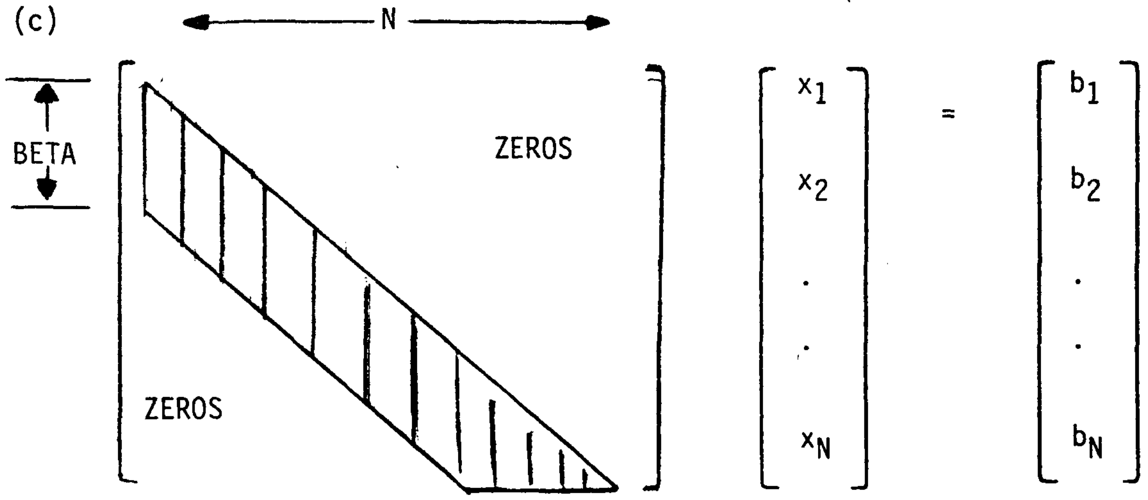
4. Iverson, K. E.: "A Programming Language," John Wiley and Sons, Inc., 1962.

## APPENDIX 1

```
(a) /*  This program segment forms the product of two matrices by repeated
        column multiplications rather than using inner products.  The vector
        array  A  is used to store the columns of a 5 x 3 matrix, B holds the
        columns of a 3 x 4 matrix and  C  will be used to hold columns of
        the product. */
    REAL VECTOR [1..5] ARRAY  (1..3)  A;
    REAL VECTOR [1..3] ARRAY  (1..4)  B;
    REAL VECTOR [1..5] ARRAY  (1..4)  C;
    /*  Assume  A  and  B  are initialized. */
    FOR  J  FROM  1 TO 3 DO
         C(J) := 0 ;
         FOR  I  FROM  1 TO 3 DO
              C(J) := C(J) + A(I)*B(J)[I] ;
         ENDF;
    ENDF;
```

```
(b) /*  This program segment solves the system AX = B where  A  is lower
        triangular.  The vector array  A  holds the rows of a lower triangular
        matrix, while the vectors  X  and  B  are column vectors. */
    REAL VECTOR [1..1 by 1] ARRAY (1..100) A;
    REAL VECTOR [1..100] X,B;
    /*  Assume  A  and  B  are initialized */
    X[1] := B[1]/A(1)[1];
    FOR  I  FROM  2 TO 100 DO
         X[I] :=(B[I] - (X[1!I-1] .DOT. A(I)[1!I-1]))/A(I)[I];
    ENDF;
```

(c)



/* This piece of code solves a unit lower triangular system LX = B as shown in the diagram. The vector array L contains the N column vectors of the matrix and each vector is of length BETA. The solution is formed in the vector B. */

```
REAL VECTOR [BETA] ARRAY (N) L;
REAL VECTOR [N] B;
REAL VECTOR [BETA] T;
FOR J FROM 1 TO N-BETA DO
     T :=B[J] * L(J)[2..BETA+1] ;
     B[J=1..J+BETA] := B[J=1..J+BETA] - T ;
ENDF;
FOR J FROM N-BETA+1 TO N-1 DO
     T[1..N-J] := B[J] * L(J)[2..N-J+1] ;
     B[J+1..N] := B[J+1..N] - T[1..N-J] ;
ENDF;
```

## APPENDIX 2

Partial list of SL/1 vector operators:

| | |
|---|---|
| FLOOR ( ) | Element by element floor of a vector. |
| CEIL( ) | Element by element ceiling of a vector. |
| SQRT( ) | Element by element square root. |
| REV( ) | Reverse a vector. |
| SUM( ) | Sum of a vector's elements. |
| PRD( ) | Product of a vector's elements. |
| MAX( ) | Maximum element of a vector. |
| MIN( ) | Minimum element of a vector. |
| < | |
| <= | |
| = | Element by element relational operators. |
| >= | The result is a bit vector. |
| > | |
| + | Element by element addition. |
| - | Element by element subtraction. |
| * | Element by element multiplication. |
| / | Element by element division. |
| ** | Element by element exponentiation. |
| .DIV. | Element by element integer division. |
| .MOD. | Element by element modulus. |
| .CON. | Concatenate two vectors. |
| .REP. | Repeat a vector. |
| .DOT. | Vector dot product. |
| .EVL. | Evaluate a polynomial |
| .AVG. | Element by Element average. |
| .CMP. | Compress a vector according to a bit vector. |